Logical time at work: capturing data dependencies and platform constraints

Calin Glitia
INRIA Sophia Antipolis Méditerranée
Team-project AOSTE, I3S/INRIA
Sophia Antipolis, France
Email: Calin.Glitia@sophia.inria.fr

Julien DeAntoni, Frédéric Mallet
Université de Nice Sophia Antipolis
Team-project AOSTE, I3S/INRIA
Sophia Antipolis, France
Email: firstname.surname@sophia.inria.fr

Abstract

Data-flow models are convenient to represent signal processing systems. They precisely reflect the data-dependencies and numerous algorithms exist to compute a static schedule that optimizes a given criterion especially for parallel implementations. Once deployed the data-flow models must be refined with constraints imposed by the environment and the execution platform. In this paper, we show how we can model data dependencies supported by multi-dimensional synchronous data flow with logical time and extend these data dependencies with additional logical constraints imposed by the environment. Making explicit these external constraints allows the exploration of further solutions during the scheduling computation.

Keywords

Logical time; synchronous data flow; CCSL

I. Introduction

Domain-specific modeling languages (DSML), just like languages, are defined by their syntax (given by a metamodel) and their (behavioral) semantics. When several DSMLs are to be integrated in the same design environment, the semantics must be explicit within the model. The Clock Constraint Specification Language (CCSL) is a DSML devised to build semantic models that shall be combined with syntactic models. The semantic model gives the behavioral interpretation and thus makes the syntactic models executable. CCSL provides constructs to describe causal and chronological relationships between model elements. Since, CCSL addresses both timed and purely causal (untimed) models, it relies on a relaxed form of time that covers both aspects, namely *logical time*.

In a previous work [1], we have shown how logical time and CCSL can be used to model the causal relationships underlying a well-known data-flow model called Synchronous Data Flow (SDF) [2]. The purpose was to execute a UML activity diagram (or any other diagram that graphically/syntactically resembles an SDF graph) with the exact execution semantics imposed by SDF. In this paper, we discuss the benefits of using a data-oriented language (like SDF) jointly with a control-oriented language (like CCSL). We also extend the approach to multi-dimensional SDF [3], for which the benefits of combining two such languages appear more clearly. Process networks in general [2]–[5] are very convenient to model signal processing systems. Several algorithms [6]–[8] exist to compute a static schedule for such models. The data-flow graphs expose the potential concurrency inherent in algorithms and whether it is authorized by data dependencies or not. When the algorithm is deployed, the environment and the platform impose new constraints that depend on the effective parallelism available. Logical time and therefore CCSL are well-adapted to capture these constraints from the platform and the environment.

In SDF the encoding of data dependencies directly leads to the description of the possible system execution schedules. However, other data flow models like Array-OL [5] describe data dependencies from which no schedules can be directly deduced. A first refinement is then necessary to translate the data dependencies into execution dependencies. In this paper, the first contribution is to explain how this translation can be done by using constrained logical time. The proposed translation does not restrict the original problem space. This is first illustrated on SDF and then on Multidimensional-SDF (MDSDF) [3].

MDSDF extends SDF to several dimensions and defines data dependencies along each dimension and partially along the multidimensional space, thus can lead to very different results depending on the synchronization between the dimensions. These synchronizations are usually computed to minimize the data accumulation in the system regardless of the environment (*e.g.,* platform, sensors) [6]. However, these synchronizations come from external constraints, which depend on the way data are collected from the environment and from the other hardware resources. A second contribution of this paper is to show how a system defined by constrained logical time can be refined with external constraints to specify the problem space resulting after deployment.

Using such an approach provides a golden model from which analysis, static scheduling, state space exploration can be driven. Moreover, it represents all the possible schedules of a system with regards to the external constraints and can be simulated in TimeSquare [9] , the tool associated with CCSL.

The next section introduces the mandatory notions about process networks semantics and CCSL. Then, the use of constrained logical time is presented in the encoding of the SDF data dependencies. The translation of this semantics into execution dependencies is described in section III-B2. After extending the same approach to MDSDF, we detail our proposition to describe external constraints. A discussion and a conclusion follow.

II. Background

A. CCSL *in a nutshell*

The Clock Constraint Specification Language (CCSL) was introduced as a companion language of the UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [10]. It provides a set of often-used time patterns that can express causal and chronological relationships between events occurring within a model. Any event can be considered as a clock, the instants of which represent the occurrences of the event. Clocks are possibly infinite and possibly dense totally ordered sets of instants. Clock instants are, at the same time, the observation of some event occurrences and model elements that dictate when events can or cannot occur. When the set of instants is discrete (discrete clocks), which is always the case here, we can sort the instants according to their chronological occurrences and we can identify each instant by an index ($i \in \mathbb{N}^\star$). $c[i]$ denotes the one instant of the clock c so that exactly $i-1$ instants of c have occurred before it. Full details about the syntax and the semantics of CCSL are available as a research report [11]. We only recall here the main relations used in the following.

CCSL relies on three main instant relations: *precedence* ($\prec$), *coincidence* ($\equiv$), *exclusion* (#). Clock relations are practical constructs to apply infinitely many instant relations between two clocks. Two clock relations are used in the following. Two clocks $c1$ and $c2$ are said to be synchronous ($c1 \boxed{=} c2$) when all their instants are coincident pair-wise: $(\forall i \in \mathbb{N}^\star)(c1[i] \equiv c2[i])$. One clock $c1$ precedes another clock $c2$ ($c1 \boxed{\prec} c2$) when $(\forall i \in \mathbb{N}^\star)(c1[i] \prec c2[i])$. Clock expressions build a set of new clocks from existing ones. Clock expression *filteredBy* (denoted $\blacktriangledown$) synchronously filters out some instants of one clock according to a static infinite periodic binary word. $a \blacktriangledown (1.0^n)^\omega$ builds a new clock b, so that $(\forall i \in \mathbb{N}^\star)(b[i] \equiv a[i*(n+1)-n])$. Clock expression *delay* (denoted \$) delays all instants of a given clock. $a \,\$\, n$ builds a new clock b, so that $(\forall i \in \mathbb{N}^\star)(b[i] \equiv a[i+n])$ and is strictly equivalent to $a \blacktriangledown 0^n(1)^\omega$.

B. *Process networks semantics*

Khan Process Networks [4] is a common model for describing signal processing systems where infinite streams of data (unbounded FIFO channels) are incrementally transformed by processes executing in sequence or parallel.

The global execution semantics of such systems is given by the set of local data dependencies between the processes (defined by the channel connections). This rule specifies that a process can only execute when his input channels contain enough data items.

These local dependencies can be defined with CCSL by associating a *logical clock* to each process execution event and by translating each local data dependency into clock constraint rules. The rules would specify that, on a channel, the read of a data element (by the *slave* process[1]) must be preceded by the write of this data element (by the *master* process).

A common application of the process networks, the data-flow languages, use a component-based approach for specifying the functionality of a system. "Actors" (or components) are the main entities. An actor consumes a fixed[2] amount of data ("tokens") from its input ports and produces a fixed amount of data on its output ports. A system is successively and hierarchically decomposed into a series of actors that are connected through data paths ("arcs"), which specify the *flow of data* in the system.

Such basic assumptions favor static analysis techniques to compute a static schedule that optimizes a given criterion (*e.g.,* the buffer sizes) but limit the expressiveness of the specification. Additional features were introduced in many derivative languages to overcome these limitations. Several data-flow specification models have been proposed throughout the time. Most of these languages were designed around the *Synchronous Data Flow* (SDF) [7], proposed by Lee and Messerschmitt, or its multidimensional extension, *Multidimensional Synchronous Data Flow* (MDSDF) [6], designed to preserve the static properties for efficient implementations, while extending its expressiveness to cover a larger range of applications.

The multidimensional extension is essential for specifying complex data-flow applications where the data structures are commonly represented as multidimensional arrays or streams. On the other hand, this extension has an important impact on the actual execution order. Whereas the SDF model defines a strict ordering in time, MDSDF only defines a partial ordering: each dimension defines quasi-independent relations "past-future", as called in [6]. External constraints need to be introduced into the system to define a complete ordering in time. With MDSDF these additional constraints are hidden in the computation of a specific schedule optimized according to a specific criterion (*e.g.,* minimizing the buffer sizes, exploit maximum of parallelism).

ARRAY-OL [5] takes the concept of multidimensional order even further, by completely mixing space and time into the data-structures at the specification level: single assignment of multidimensional arrays with possibly infinite dimensions (maximum

[1]The direction of the channel defines the relation *master-slave* between the two processes at its ends.

[2]Numerical values known at specification time

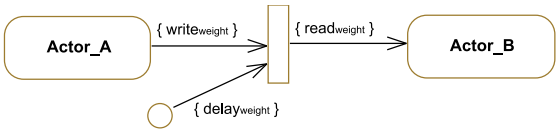

Fig. 1. SDF arc

one by array) define the set of data values that will transit through the system. Data dependencies between uniform and repetitive patterns of data are defined. The global order depends on the sets of depending pairs of actor executions, where two actor instances are execution-dependent if the patterns produced/consumed share at least a common data element. In such a MoCC, a total order between executions cannot be deduced unless additional environment constraints are specified.

III. Synchronous data-flow

We briefly recall in Section III-A our previous proposition [1] to define with CCSL the execution semantics of the Synchronous Data Flow. This proposition purely based on data-dependencies is then refined into a new proposition (Section III-B), where local scheduling decisions are directly expressed in CCSL. It is important to note that such decisions are only local and does not reflect the global scheduling.

A. Semantics based on data dependency

Our initial proposition was to translate the data dependencies defined by arcs into CCSL relations. The actor executions are modeled by logical clocks. Each instant of the clock denotes one execution of the related actor. Logical clocks are also used to model read/write operations on the arcs. The CCSL rule associated to an arc (Figure 1) represents a conjunction of three relations, as follows:

1) A packet-based precedence on the inputs states that *weight* read events from the arc are needed before an actor can executed. The strictly positive integer *weight* represents the input weight:
$$\left(read \blacktriangledown \left(0^{weight-1}.1\right)^{\omega}\right) \boxed{\prec} actor$$
2) Each actor execution is followed by *weight* write events on the output arcs, where the strictly positive integer *weight* represents the output weight:
$$actor \boxed{=} \left(write \blacktriangledown \left(1.0^{weight-1}\right)^{\omega}\right)$$
3) For a given arc, the i^{th} tick of *write* must precede the i^{th} tick of *read*: $write \boxed{\prec} read$. When *delay* tokens are initially available in the queue, the i^{th} read operation uses the data written at the $(i-delay)^{th}$ write operation, for $i > delay$:
$$write \boxed{\prec} (read \$ delay)$$

The data dependencies between two actors at the ends of an arc are expressed in this proposition by CCSL clock constraints between element-wise production/consumption on this arc. For SDF models with actors that produce and consume a larger number of tokens by execution, this approach explodes in size at simulation. Moreover, the essential aspects between the relative execution of actors would be completely negligible compared to the overwhelming information concerning token writings and readings.

B. Semantics based on execution dependency

Therefore, we propose a new way to translate data dependencies induced by a SDF arc into CCSL relations between actor executions, without going through the element-wise write/read operations. The read tokens/execute actor/write tokens operations are abstracted by a single atomic event.

1) Encoding the local scheduling in CCSL*:* Expressing the execution dependency between a master and a slave actor (linked by an arc) means identifying the minimum number of executions of the master actor needed to enable the slave actor execution. Lets first consider the trivial case of an arc with no initial delays and where the master actor produces an arbitrary number t of token, while the slave actor consumes twice as many tokens $(2 \times t)$ for each execution. The minimum number of executions of the master actor that enables the slave actor is 2. It can be expressed with CCSL as
$$\left(Clock_{master} \blacktriangledown (01)^{\omega}\right) \boxed{\prec} \left(Clock_{slave} \blacktriangledown (1)^{\omega}\right) \quad (1)$$

Operator *filteredBy*(▼) allows to filter the clock events according to a binary periodic word. The left binary word $(01)^{\omega}$ filters all the odd executions of the master while the right binary word $(1)^{\omega}$ keeps all the slave executions. These patterns encode the minimum number of executions of the master actor to enable the slave actor execution (*i.e.,* 2). The pattern identifies the exact

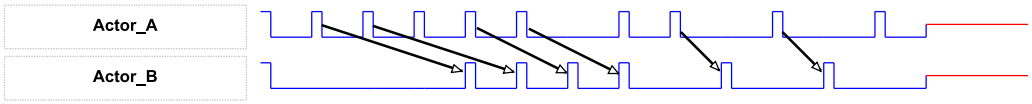

Fig. 2. SDF arc as a precedence relation

TABLE I
COMPUTING THE **FILTEREDBY** BINARY WORD.

	initial		+4	+4	-6+4	-6+4
tokens	1		5	9	7	5
			< 6	> 6	> 6	done
binary		(	0	1	1	)

pairs of depending actor executions: each pair is formed by: 1) the master actor execution that produces the data elements (token) which enable the slave and 2) the corresponding slave actor execution.

In the general case, these patterns can be more complex; replacing $write_{\text{weight}} = 4$ $read_{\text{weight}} = 6$ and $delay_{\text{weight}} = 7$ on Figure 1 defines the clock relations on Figure 2. As the production rate does not exactly divide the consumption rate, the computation of a local scheduling is needed to obtain the patterns, which represent the exact pairs of depending actor executions.

The CCSL relation that defines the dependencies between the executions of two actors connected by an arc (as in Figure 1) has the following general form:

$$(Clock_{master} \blacktriangledown 0_bM) \boxed{\prec} ((Clock_{slave} \$ d) \blacktriangledown 0_bS) \tag{2}$$

where 0_bM, 0_bS represent two binary periodic words and d a positive integer. These values are computed from the parameters of the arc: production rate ($write_{\text{weight}}$), consumption rate ($read_{\text{weight}}$) and initial delays ($delay_{\text{weight}}$). Eq. 2 expresses the exact data dependencies between the executions of the two actors at the two ends of an arc, as follows:

- If enough tokens are initially available on the arc, the slave actor can execute d times without waiting the execution of the master actor, where $d = \lfloor delay_{\text{weight}}/read_{\text{weight}} \rfloor$.
- If the production rate is lower than the consumption rate ($write_{\text{weight}} < read_{\text{weight}}$), several executions of the master actor are needed to produce the tokens consumed by one execution of the slave actor. This relation is expressed by the *filteredBy* relation on the left of Eq. 2.
- If the production rate is higher than the consumption rate ($write_{\text{weight}} > read_{\text{weight}}$), each execution of the master actor produces more tokens than consumed by a single execution of the slave actor and therefore sooner or later the accumulated tokens will allow the execution of the slave actor multiple times. This relation is expressed as the *filteredBy* relation on the right of Eq. 2.

The parameters for the *filteredBy* relations represent periodic binary words that exclusively depend on the parameters of the arc and can be computed using an iterative algorithm introduced below. Parts of the relation can be omitted as expressions like *delayedFor* 0 or *filteredBy* $(1)^\omega$ have no actual effect.

2) Local scheduling algorithm: An iterative algorithm can be used to compute the binary word of a *filteredBy* relation and represent the computation of a local As Soon As Possible (ASAP) scheduling between the two actors.

We only treat the case where the production rate is lower than the consumption rate. The opposite case leads to a similar algorithm.

1) Starting with the initial tokens on the arc ($initial = delay_{\text{weight}}$ mod $read_{\text{weight}}$), at each step we test if there are enough tokens to be consumed or not.
2) If true, the slave actor can execute and the consumed tokens are removed from the arcs, while the value 1 is added to the binary word. If false, the value 0 is added to the binary word.
3) At each step, the produced tokens are added to the arc.
4) This iteration stops when the number of tokens on the arc represents a value already processed, case when the periodic value is reached.
5) *Observation.* The algorithm will stop after maximum $read_{\text{weight}}$ steps, as at each step the number of tokens is less that $read_{\text{weight}}$ and it stops when it reaches a value already processed.

a) Example: The computation of the CCSL relation for the previous example is illustrated in the Table I. The $delay_{\text{weight}}$ of 7 tokens allows the slave actor to execute once ($d = \lfloor 7/6 \rfloor = 1$) independently from the execution of the master clock, which leaves one initial token that will be used to compute the binary word for the *filteredBy* relation of $0_b(011)$, according to our algorithm.

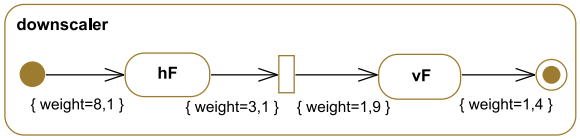

Fig. 3. Downscaler

Eq. 2 becomes:

$$\left(Clock_{master} \blacktriangledown (011)^{\omega}\right) \prec \left(Clock_{slave} \$ 1\right) \tag{3}$$

and gives a possible execution as the one shown on Figure 2.

Such algorithm gives an ASAP scheduling between the two actors at a local level but the *precedence* ($\prec$) relation defines the minimal execution dependency between the two actors so that any scheduling that respects the relation is correct. The set of local relations defines the global execution dependencies between all the actors in the system and specifies its global behavior.

C. Semantics Comparison

The semantic definition should express correctly and completely the model considered. As a Process Network defines the global behavior as the sum of the local rules, having a correct and complete translation of a local rule into CCSL relations guarantees the correctness of the semantics at the global level. In both presented approach, only local rules have been encoded.

For both of the presented SDF semantic definitions, the clock constraint rules between the actor execution events are equivalent but represent two different levels of abstraction. The semantics definition based on data dependency expresses these dependencies as element-wise precedence relations between the write and the read of a data element, while the semantics definition based on execution dependency abstracts the manipulation of the data elements by a local schedule for each arc that focuses on the actor executions. The second approach avoid the overhead that occurs when large amount of data are read or written for each actor execution.

A correct definition of the semantics of a MoCC must ensure it does not restrict the original problem space. The dependencies defined in a MoCC should translate into minimal but sufficient CCSL rules and in this way define a system behavior encoding the entire range of correct schedules.

IV. Extensions to Multidimensional Data-Flow

A. Semantics

Multidimensional SDF (MDSDF) was introduced to provide ways to express the number of tokens produced and consumed in a stream with more than one dimension. The principles of MDSDF are quite similar to those of SDF. On each task we just have to specify the number of data consumed and produced on each dimension.

B. Encoding MDSDF in CCSL

The multidimensional SDF model is a straightforward extension of 1-D SDF. The number of tokens produced and consumed are now given as M-tuples, where M represents the number of dimensions of the model. M represents the maximum number of dimensions in the model; tuples with a number of dimensions inferior to M have by default the last values equal to 1.

Instead of one balanced equation for each arc, there are now M relations producer/consumer, one for each dimension. This can also be expressed straightforwardly from the semantics of SDF; it is sufficient to declare M clocks for each actor executions (one for each dimension) and to translate the balanced equation for each arc and for each dimension into a precedence relations (Eq. 2).

Independent balanced equations for each dimension allow this multidimensional model to remain compatible with the mono-dimensional SDF, but limit the expressiveness of the accesses to continuous rectangular blocks. Furthermore, relations between actor executions are expressed at a global level as a set of SDF-like and quasi-independent systems, one for each dimension, making it difficult to express relations throughout multiple dimensions of the model. Actor executions are projected into the multidimensional space, where the actual number of executions at an arbitrary moment is given by the multiplication of all these projections. For the semantic implementation in CCSL, an actor has a different clock for each dimension that expresses the evolution of this actor in this dimension. The actual number of actor executions at a given instant can be computed by the multiplication of the clock tick indexes of all its dimensions. Each time a clock ticks on one dimension it implies multiple executions of the actor, depending on the states of the clocks on the other dimensions.

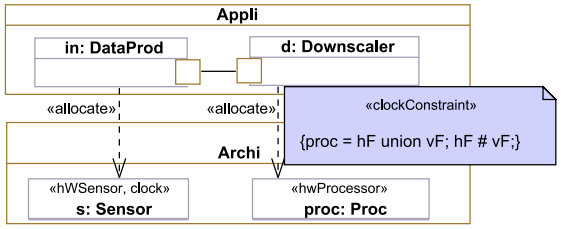

Fig. 4. External constraints

Figure 3 shows a simple application that downscales a 2D image by applying two successive filters, a horizontal filter *hF* taking 8 pixels on the first dimension and producing 3 and a vertical filter *vF* taking 5 pixels on the second dimension and producing 2. The CCSL relations encoding these data dependencies are:

$$\begin{array}{ll} in_1 \prec \left(hF_1 \blacktriangledown (1.0^2)^{\omega}\right) & hF_1 \prec \left(vF_1 \blacktriangledown (1.0^2)^{\omega}\right) \\ in_2 \prec hF_2 & \left(hF_2 \blacktriangledown (0^4.1)^{\omega}\right) \prec vF_2 \end{array} \tag{4}$$

as two sets of independent relations (one par dimension).

Data dependencies for MDSDF specify only partial execution dependencies on the multidimensional system. The effective actor execution events appear only after the computation of the static scheduling by a specific algorithm, at compilation time [6]. At this stage, the functional specification is further constrained by the execution or the environment, what we consider as external constraints. Rather that hiding these constraints within the compilation stages, we consider that these constraints should be explicitly expressed within the specification, as additional CCSL constraints in our particular case.

In Section V, we discuss how external constraints defined by the execution platform or the environment can be used to refine the system by translation into additional clock relations.

V. External constraints

We have seen how the semantics of a model can be defined using the CCSL language. Such semantics can be associated to any other syntactic model. The semantic model can then be used to validate certain aspects or properties of the system, through formal verification on the CCSL language or simulation by using tools like *TimeSquare*.

Like in the examples presented in Section III and IV, the data dependencies define the internal constraints of a system. External constraints that come from the environment or the execution platform should be explicitly considered and used to refine the system. Since these extensions are not dictated by the targeted language, they should not modify the constraints, just enforce them. Later, such relations are used to extend the CCSL specification depending, for instance, on the chosen allocation to the execution platform. It results in an explicit characterization of the set of acceptable schedules that conforms the execution semantics and satisfies the constraints from the execution platform.

A first interesting aspect consists in using an information representing the capacity of a buffer and to use an algorithm similar to the one proposed in Section III-B2 to constrain the memory size of one or several buffers. The constraint is computed from a platform information and not hard-coded in the model by a back-pressure arc (contrary to [1]).

Another illustration that highlights the benefits of using a data-flow language like MDSDF conjointly with a control-oriented language appears when addressing the concept of multidimensional ordering discussed in the context of multidimensional data structures. For the mono-dimensional data flows of SDF, the data follows a complete order, while for MDSDF only a partial order is defined: no ordering between different dimensions is defined. Such an ordering, rather than being computed by an algorithm independently of the execution platform, may be enforced by the way the data are collected in the execution platform or its environment. For instance, in a video processing application, depending on the used sensors, the 2D video input enters the system as a flow of pixels, line by line or even image by image. Like previously, we propose to specify such external constraints in CCSL. By defining an ordering on the input data flows, the external constraints introduce constraints across the dimensions and consequently impact the set of acceptable schedules. The order on the data entering the system is propagated throughout the system, defining a global order between the dimensional components of each actor clocks and allows the definition of unified clocks encoding the relative multidimensional order.

For the downscaler application on Figure 3, the external constraints define the order in witch data enters the system (by rows of 24 pixels for this example) and execution constraints specifying that the two filters are executed on the same processor *proc* (Figure 4). The order of data entering the system (sensor clock *s*) defines a relative order between the two dimensions

and actual actor execution clocks:

$$\begin{array}{ll} in_1 = \left(s \blacktriangledown 1.(0)^{\omega}\right) & in_2 = s \\ hF_1 = \left(hF \blacktriangledown 1^3.(0)^{\omega}\right) & hF_2 = \left(hF \blacktriangledown \left(0^2.1\right)^{\omega}\right) \\ vF_1 = \left(vF \blacktriangledown 1^9.(0)^{\omega}\right) & vF_2 = \left(vF \blacktriangledown \left(1.0^8\right)^{\omega}\right) \end{array} \tag{5}$$

A last interesting example of external constraints considers the deployment of a data-flow model onto an execution platform. By nature, data flow models can be massively concurrent. However, the functional concurrency is restricted by the actual physical parallelism of the execution platform. This is achieved by specifying a clock for each processor. A processor clock is defined by the *union* of all the actor clocks that are allocated to this processor. Moreover, because actors allocated to a same processor can not be concurrent, their clock are in an *exclusion* relation ($\#$), as those defined for the downscaler example:

$$proc = hF + vF \qquad hF \# vF \tag{6}$$

VI. Discussion and conclusion

CCSL was initially developed as a companion language for the time subprofile of the UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [10]. It has developed ever since independently of the UML as a language based on logical time to define a timed causality semantics of models. Syntactic model are complemented with a semantic model described in CCSL. The behavior of a system is thus expressed as a formal specification encoding the set of schedules corresponding to a correct execution. In this paper, we specifically focus on data flow languages. The selected languages have the particularity to be compositional. The semantics of the system is fully defined by the set of local execution rules imposed by the data dependencies.

A first semantics definition that encodes this fine grain element-wise data dependencies for SDF was previously proposed in [1]. Our first contribution is to propose an algorithm when execution dependencies between the actor activation events are directly expressed by translating the local data dependencies. An algorithm corresponding to the computation of a local scheduling for each pair of dependent actors is proposed. This computation stage has just the role of translating the data dependencies into execution dependencies and does not restrict in any way the problem space. For more complex languages, translating the data dependencies into execution dependencies that can be expressed by the CCSL language implies more complex computations. It is the case of Array-OL or other polyhedral models where data dependencies are defined relatively to regular but arbitrary shaped sub-arrays. Defining in CCSL the definition for Array-OL is a complex problem that shall be fully discussed in a future work.

As we have shown, the behavior definition of SDF was extended straightforwardly to the multidimensional space of MDSDF, as a set of SDF-like independent systems, one for each dimension. In this context, we encountered the problem of multidimensional ordering which is just partially defined by MDSDF, while a complete order is required when performing the static scheduling. CCSL is used to reduce the possible ordering and then make explicit this choice otherwise hidden in the chosen scheduling algorithm. In our point of view, these scheduling "choices" should be expressed within the system specification as external constraints, *i.e.,* as execution platform constraints defining the multidimensional ordering of the data in inputs of the system. Other execution platform constraints are also addressed, as the available computational resources (parallel processors), the storage resources (buffer sizes), etc. These external constraints are parametrized by a specific allocation of the model onto the platform. They are then juxtaposed with the other language-specific constraints. This ensures that the platform constraints are a refinement but does not altered the initial specification.

The translation from data dependencies to execution dependencies, the automatic application of SDF and MDSDF execution semantics onto UML activity diagram as well as the addition of platform constraints are already available as an experimental feature of TimeSquare (available for download).

References

[1] F. Mallet, J. DeAntoni, C. André, and R. de Simone, "The clock constraint specification language for building timed causality models," *Innovations in Systems and Software Engineering*, vol. 6, no. 1–2, pp. 99–106, 2010.

[2] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[3] E. A. Lee, "Mulitdimensional streams rooted in dataflow," in *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, ser. IFIP Transactions, M. Cosnard, K. Ebcioglu, and J.-L. Gaudiot, Eds., vol. A-23. North-Holland, 1993, pp. 295–306.

[4] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing*, pp. 471–475, 1974.

[5] C. Glitia, P. Dumont, and P. Boulet, "Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing," *Multidimensional Systems and Signal Processing*, vol. 21, no. 2, pp. 105–131, 2009.

[6] M. J. Chen and E. A. Lee, "Design and implementation of a multidimensional synchronous dataflow environment," in *Proc. IEEE Asilomar Conf. on Signal, Systems, and Computers*, 1995.

[7] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[8] P. Dumont and P. Boulet, "Another multidimensional synchronous dataflow: Simulating Array-OL in ptolemy II," Tech. Rep. RR-5516, Mar. 2005. [Online]. Available: http://www.inria.fr/rrrt/rr-5516.html

[9] Projet INRIA AOSTE, "TimeSquare," http://www-sop.inria.fr/aoste/dev/time_square/.
[10] OMG, *UML Profile for MARTE, v1.0*, Object Management Group, Nov. 2009, document number: formal/09-11-02.
[11] C. André, "Syntax and Semantics of the Clock Constraint Specification Language (CCSL)," INRIA, Research Report RR-6925, 2009. [Online]. Available: http://hal.inria.fr/inria-00384077/en/